LA-UR-21-26924
Approved for public release; distribution is unlimited.

Title: Evaluating Symmetry in 3-dimensional Simulation Data

Author(s): Woods, Charles Nathan

Intended for: Report

Issued: 2021-07-20 (rev.1)

Evaluating Symmetry in 3-dimensional Simulation Data

C. Nathan Woods, XCP-8

Introduction

The increasing use of three-dimensional simulations has led to an increased need for effective verification and validation, or V&V. Unfortunately, the high computational cost associated with three-dimensional simulations makes many standard V&V techniques cost-prohibitive. Given these circumstances, we must pay close attention to V&V metrics that can be evaluated using only a single simulation. One of the simplest of these metrics is the evaluation of physical symmetries. This report details a method of computing the deviation from symmetry in a three-dimensional dataset with an arbitrarily distributed mesh. Examples are presented for spherical and axisymmetric systems using the Paraview post-processing analysis workflow.

Overview

The essential idea of this algorithm is to compare a field value at a given point or cell against the average value of that field among all points or cells that ought to be symmetric. We will assume that our data is defined on a mesh that is not exactly symmetric, and that cell quantities are defined at cell centroids. The lack of an aligned grid is the principal challenge to this analysis, and we show how to overcome it.

Averaging 3D Data over Surfaces of Symmetry

Suppose that we have a field $\phi: M \to \mathbb{R}^n$ and we wish to average it over some surface $S \subset M$. The field ϕ is represented as a set of scalar or vector values at discrete points (denoted by ϕ_i) which do not necessarily coincide with the surface S. Using continuous functions and variables, we would compute the average value of ϕ over S by integration:

$$\bar{\phi} \equiv \iint_S \phi \, dS \Big/ \iint_S dS.$$

Since we do not assume that we have a set of discrete facets that aligns with the grid, we must find some way to approximate this integral. We do not choose to interpolate grid values onto the aligned surface. Instead, we will perform a weighted integration, with the weight function chosen to include only those values which ought to affect the average. The only discrete numerical operation we will need is a volume integral:

$$\iiint_V \phi \, dV \approx \sum_i \phi_i \, \Delta V_i$$

In choosing our weight function, we want something that varies smoothly across its extent while ignoring points that are far from the averaging surface. In other words, we want to sample our field with a C^∞-continuous function that has compact support (i.e. it is identically zero outside of a well-defined area). This class of functions is commonly called bump functions (https://en.wikipedia.org/wiki/Bump_function), and one example is:

$$\Psi(x) = \begin{cases} \exp\left(\frac{1}{(x^2-1)}\right) & ; \quad x \in (-1,1) \\ 0 & ; \quad otherwise \end{cases}$$

In our case, we want to be able to control the width of our bump, which yields:

$$\Psi(x,h) = \begin{cases} \exp\left(\frac{h^2}{(x^2-h^2)}\right) & ; \quad x \in (-h,h) \\ 0 & ; \quad otherwise \end{cases}$$

We would normally need to normalize this function for a given value of h:

$$\Psi'(x,h) \equiv \frac{\Psi(x,h)}{\int_{-h}^{h} \Psi(x,h)dx} \cong \frac{\Psi(x,h)}{0.443994\,h}$$

It is straightforward to show:

$$\lim_{h\to 0} \iiint_V \Psi'(x,h)\phi(x)dV = \iint_S \phi(S)dS$$

This normalization factor will cancel out when we compute the average, so we will neglect it.

From this, we may define:

$$\iint_S \phi\, dS \approx \sum\nolimits_i \Psi(x_i,h)\, \phi_i\, V_i$$

From this, we may compute the volume-averaged value of the field:

$$\bar{\phi}(x) \approx \frac{\sum_i \phi_i \Psi_i\, V_i}{\sum_i \Psi_i\, V_i}$$

Evaluating the surface integral in this way is mathematically equivalent to performing a smoothing convolution using the bump filter. The parameter h effectively controls the degree of the smoothing.

Example 1: Spherical Symmetry

In the case of spherical symmetry, for a field defined in coordinates that are not necessarily spherical, we are attempting to compute:

$$\bar{\phi}(r) \equiv \frac{\int_0^{2\pi}\int_0^{\pi} \phi(\vec{r})\, r^2 \sin\theta\, d\theta\, d\phi}{\int_0^{2\pi}\int_0^{\pi} r^2 \sin\theta\, d\theta\, d\phi}$$

We note that the symbol ϕ has two meanings in the above: the field to be averaged; and the azimuthal coordinate. We will approximate this integral using a weighting function:

$$\bar{\phi}(r) \approx \frac{\int_0^{\infty}\int_0^{2\pi}\int_0^{\pi} \phi(\vec{r})\, \Psi(r-r',h)\, r'^2 \sin\theta\, d\theta\, d\phi\, dr'}{\int_0^{\infty}\int_0^{2\pi}\int_0^{\pi} \Psi(r-r',h)\, r'^2 \sin\theta\, d\theta\, d\phi\, dr'}$$

The integral weighting is based on the radial coordinate r, so we must compute that for each point in the field. We are not interested in the angular dependence, so we may neglect that, yielding:

$$\bar{\phi}(r) \approx \frac{\int_0^{\infty}\int_0^{2\pi}\int_0^{\pi} \phi\big(r(\vec{r})\big)\,\Psi(r-r',h)\,r'^2\sin\theta\,d\theta\,d\phi\,dr'}{\int_0^{\infty}\int_0^{2\pi}\int_0^{\pi} \Psi(r-r',h)\,r'^2\sin\theta\,d\theta\,d\phi\,dr'}$$

$$= \frac{\int_V \phi\big(r(\vec{r})\big)\,\Psi(r-r',h)\,dV}{\int_V \Psi(r-r',h)\,dV}$$

This yields the expected average value:

$$\bar{\phi}(r) \approx \frac{\sum_i \phi_i \Psi_i V_i}{\sum_i \Psi_i V_i}$$

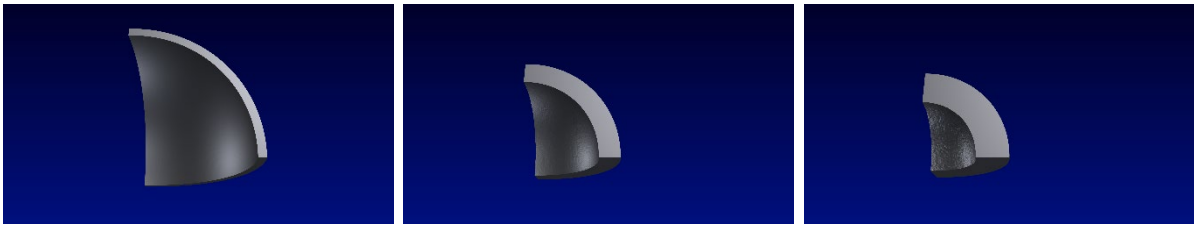

Figure 1 Collapse of spherical Verney shell, showing the motion and thickening of the metal. This was modeled as an octant under the assumption of planar symmetry, using the Fuel code. The simulation author is Vince Chiravalle.

This can be implemented in Paraview, and we demonstrate this capability using a dataset generated for the Ristra project by Vince Chiravalle using the Fuel code. In this variant of the classical Verney problem, which was developed as an approximate model for spherical metal shells collapsing under loading. The

metal shell has an initial velocity directed toward the center of the sphere, and it continues to collapse until the initial kinetic energy of the material is dissipated by plastic work. The simulation is very aggressively de-refined, and run on an octant assuming planar symmetry at the boundaries. The analysis algorithm was run on a fully spherical dataset obtained by reflection of the octant, and this is somewhat apparent in the resulting visualizations.

Our analysis shows that peak deviations from spherical symmetry are small. They are most pronounced at the coordinate axes and at the 45° angle. Because the deviations are computed relative to the average value of this dataset, it is impossible to tell from this data alone whether the code is under- or over-predicting the values as a whole. However, it seems that something is different between the handling of the planar symmetry boundaries and the rest of the simulation.

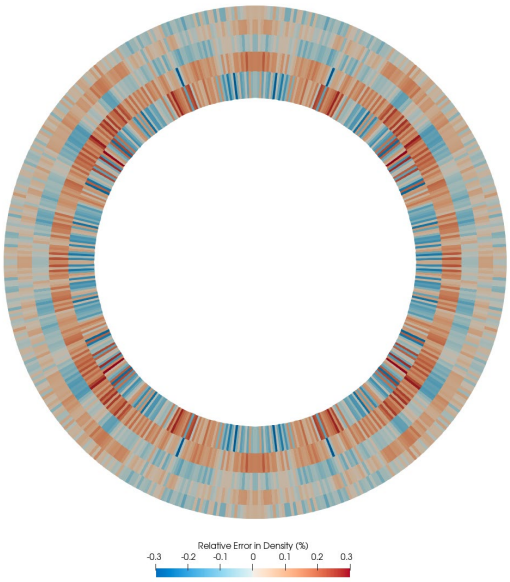

Figure 2 Relative error in density for the spherical Verney problem, taken from a slice along the planar symmetry axis. The artifical reflection of the octant dataset is apparent in the cruciform distribution of color.

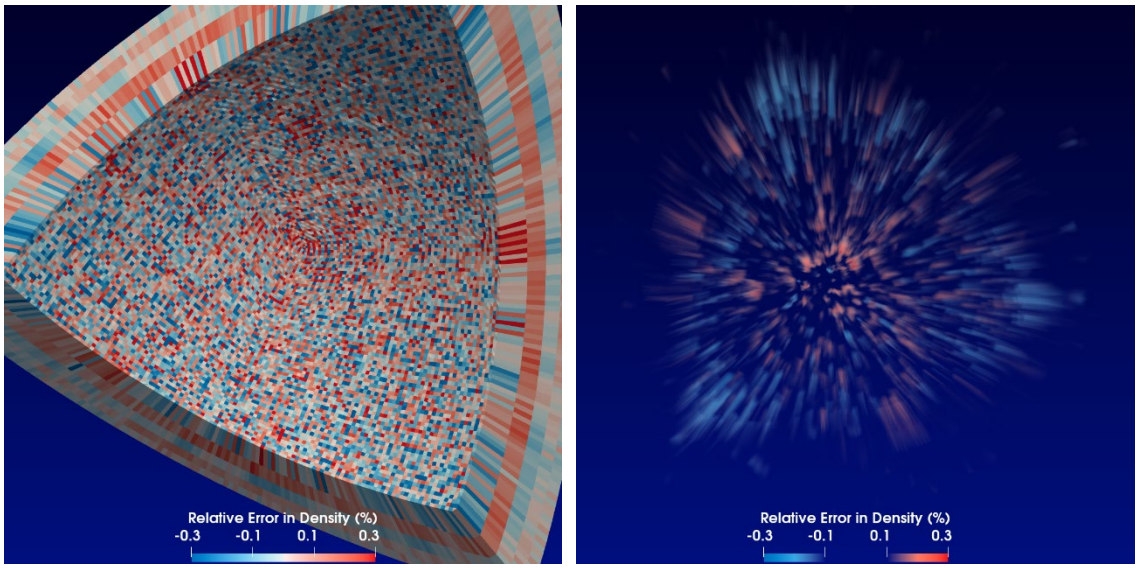

Figure 3 Surface and volumetric images of collapsing Verney shell, showing the distribution of errors in density relative to the average density value. The density is under-predicted near the planar symmetry boundary conditions, especially at the corners. Density is over-predicted in the center, near the box mesh joint. These results are relative to the average computed value; comparisons with an exact solution may yield different results.

Example 2: Cylindrical Symmetry

In the case of cylindrical symmetry (or axisymmetry), again noting that the symbol ϕ has two meanings, we are attempting to compute:

$$\bar{\phi}(r,z) \equiv \frac{\int_0^{2\pi} \phi(\vec{r})\, r\, d\phi}{\int_0^{2\pi} r\, d\phi}$$

The integral weighting is now based on both the radial coordinate r and the axial coordinate z, but we are again disinterested in the angular dependence, yielding:

$$\bar{\phi}(r,z) \approx \frac{\int_0^{2\pi} \phi\big(r(\vec{r}), z(\vec{r})\big)\, r\, d\phi}{\int_0^{2\pi} r\, d\phi}$$

The sampling becomes:

$$(\phi * \Psi)(r,z,h) = \frac{1}{2}\int_{-\infty}^{\infty}\int_{0}^{\infty} \phi(r',z')\Psi(r-r',h)\Psi(z-z',h)dr'\ dz'$$

The integration over angle is written:

$$\int_{C(r,z)} \phi(r,z,\theta)dC \approx \int_V \phi(r',z',\theta)\Psi(r'-r,h)\Psi(z'-z,h)dV \approx \sum_i \phi_i \Psi({r_i}', {z_i}', r, z, h)\, V_i$$

This yields the expected average value:

$$\bar{\phi}(r) \approx \frac{\sum_i \phi_i \Psi_i\, V_i}{\sum_i \Psi_i\, V_i}$$

We can demonstrate this capability using another dataset that Vince Chiravalle generated for the Ristra project using the Fuel code. In this simulation, a very high-pressure gas is sandwiched between two concentric, cylindrical, steel shells. The rapid expansion of the gas drives the inner shell toward the axis of symmetry, which compresses the air in the center region until the pressure is high enough to halt the motion of the steel. A quadrant of the problem was simulated in three dimensions. Our analysis shows that there is a significant deviation from symmetry at the boundaries of the quadrant, especially near the axis of symmetry.

We investigated whether the deviation in symmetry observed near the reflection planes could result from boundary effects in the analysis algorithm by using Paraview to reflect the data across those planes and do the analysis on the cylindrical data. No change was observed. Future work will examine the Fuel code to determine whether this error arises from the implementation of the planar symmetry boundary conditions.

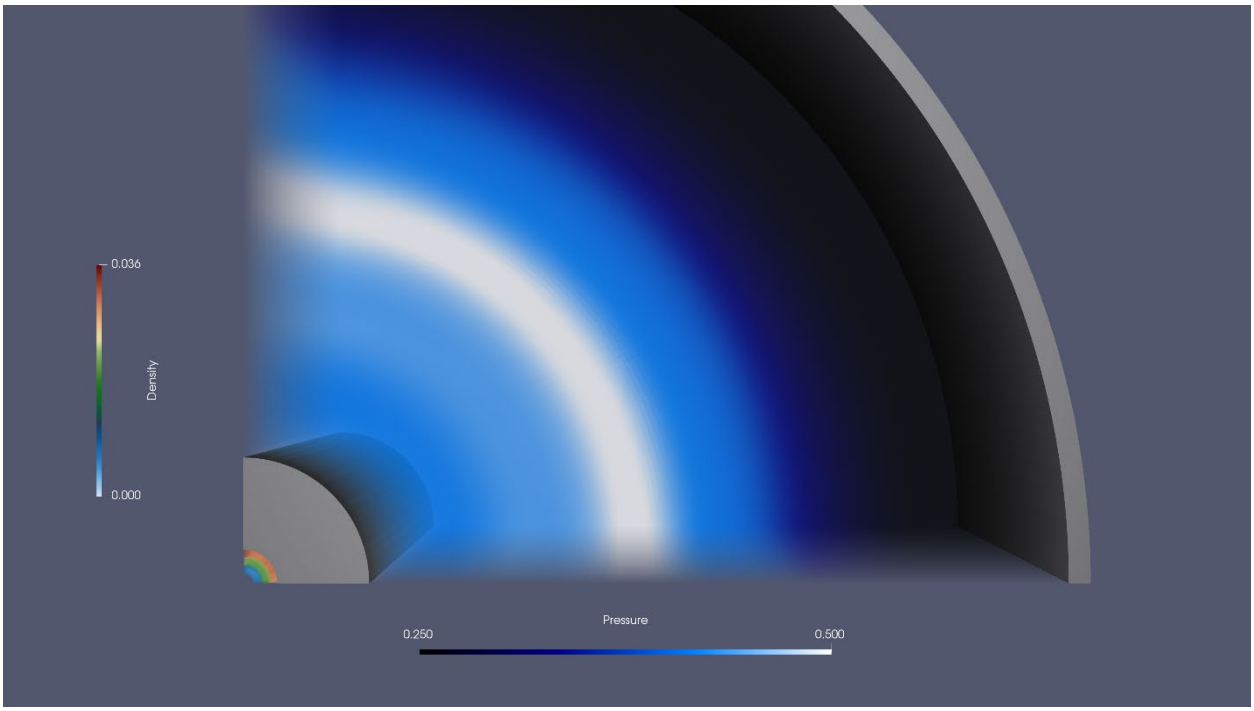

Figure 4 Late-time simulation results from the axisymmetric high-pressure gas sandwich. A high pressure gas (blue) is sandwiched between concentric steel shells (grey). The pressure of the gas drives the inner shell to implode toward the axis of symmetry until the inner gas (red, green, blue) reaches high enough pressure to halt the motion of the metal.

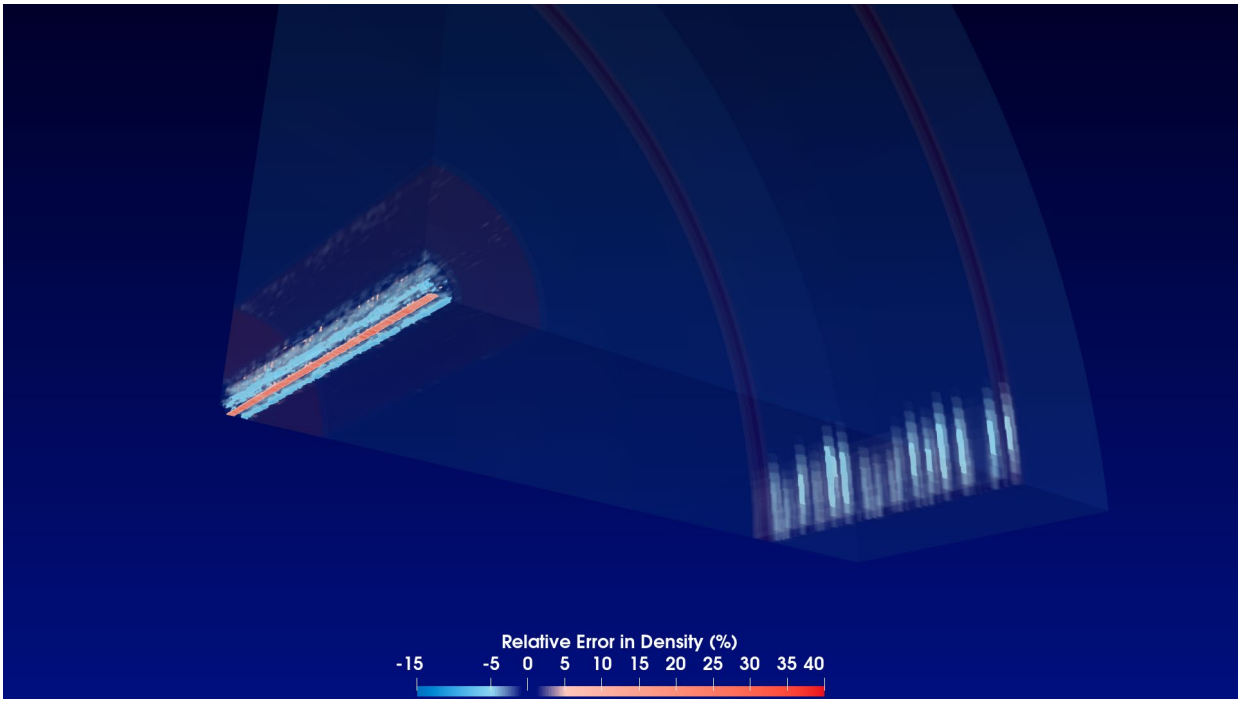

Figure 5 Relative error in density for the axisymmetric high-pressure gas sandwich. The density field is lightly overlaid in order to provide visual context, showing the metal shells in red and the gases in blue. The highest errors are found near the axis of symmetry, at the planar symmetry boundaries. Interestingly, there is also significant error at the x-axis. The corresponding location on the y-axis (not shown) has no similar error.

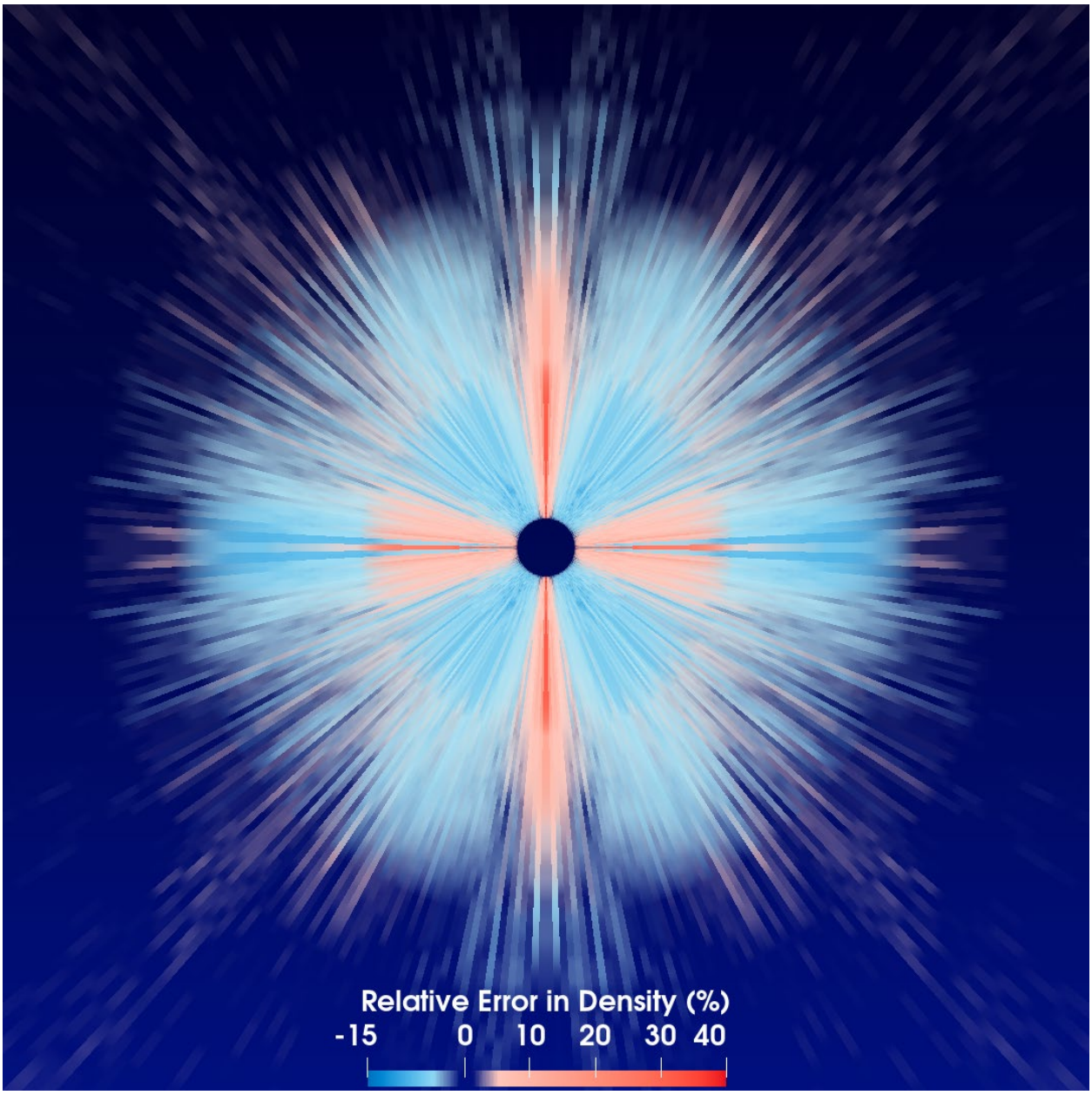

Figure 6 View down the axis of the highly converged center region of the axisymmetric high-pressure gas sandwich. The quadrant that was simulated has been reflected across the axes of symmetry. The cruciform pattern of the density errors is quite prominent.

```
# trace generated using paraview version 5.8.1
#
# To ensure correct image size when batch processing, please search
# for and uncomment the line `# renderView*.ViewSize = [*,*]`

#### import the simple module from the paraview
from paraview.simple import *
#### disable automatic camera reset on 'Show'
paraview.simple._DisableFirstRenderCameraReset() # create a new 'EnSight Reader'
engoldcase =
EnSightReader(CaseFileName='D:\\Visualization\\Ristra\\verney\\ensight\\engold.case')
engoldcase.CellArrays = ['den', 'vol']
engoldcase.PointArrays = ['vel', 'kfix']
# create a new 'Merge Blocks'
mergeBlocks1 = MergeBlocks(Input=engoldcase)

# create a new 'Ghost Cells Generator'
ghostCellsGenerator1 = GhostCellsGenerator(Input=mergeBlocks1)
ghostCellsGenerator1.MinimumNumberOfGhostLevels = 2

# create a new 'Calculator'
calculator1 = Calculator(Input=ghostCellsGenerator1)
calculator1.ResultArrayName = 'Coords'
calculator1.Function = 'coords'

# create a new 'Point Data to Cell Data'
pointDatatoCellData1 = PointDatatoCellData(Input=calculator1)
pointDatatoCellData1.ProcessAllArrays = 0
pointDatatoCellData1.PointDataArraytoprocess = ['Coords']

# create a new 'Calculator'
calculator2 = Calculator(Input=pointDatatoCellData1)
calculator2.AttributeType = 'Cell Data'
calculator2.ResultArrayName = 'CoordsR'
calculator2.Function = 'mag(Coords)'

# create a new 'Programmable Filter'
programmableFilter1 = ProgrammableFilter(Input=calculator2)
programmableFilter1.Script = """
r = inputs[0].CellData['CoordsR']
v = inputs[0].CellData['vol']
m = v * inputs[0].CellData['den']
h = .1
h2 = h**2
unsorted_out = 0*r
sortargs = argsort(r)
unsortargs = argsort(sortargs)
r_sorted = r[sortargs]
m_sorted = m[sortargs]
v_sorted = v[sortargs]
for ind in range(len(r_sorted)):
  r0 = r_sorted[ind]
  min, max = searchsorted(r_sorted, (r0-h, r0+h))
  x2 = (r_sorted[min:max] - r0)**2
  # This is where most of the expense is. Perhaps a faster exp function?
  Psi = exp(h2/(x2-h2))
  # This is also expensive:
  unsorted_out[ind] = (sum((Psi*m_sorted[min:max]))/
                       sum((Psi*v_sorted[min:max])))
print(total_times)
output.CellData.append(unsorted_out[unsortargs], 'denbar')
                       """
programmableFilter1.RequestInformationScript = ''
programmableFilter1.RequestUpdateExtentScript = ''
programmableFilter1.CopyArrays = 1
programmableFilter1.PythonPath = ''

# create a new 'Calculator'
calculator3 = Calculator(Input=programmableFilter1)
calculator3.AttributeType = 'Cell Data'
calculator3.ResultArrayName = 'Relative Error in Density (%)'
calculator3.Function = '(den - denbar)/denbar * 100'
```

```
# trace generated using paraview version 5.8.1
#
# To ensure correct image size when batch processing, please search
# for and uncomment the line `# renderView*.ViewSize = [*,*]`

#### import the simple module from the paraview
from paraview.simple import *
#### disable automatic camera reset on 'Show'
paraview.simple._DisableFirstRenderCameraReset()

# create a new 'EnSight Reader'
engoldcase = EnSightReader(CaseFileName='D:\\Visualization\\Ristra\\5mat\\ensight\\engold.case')
engoldcase.CellArrays = ['vol00', 'd00', 'vol01', 'd01', 'vol02', 'd02', 'vol03', 'd03', 'vol04', 'd04']

# create a new 'Merge Blocks'
mergeBlocks1 = MergeBlocks(Input=engoldcase)

# create a new 'Ghost Cells Generator'
ghostCellsGenerator1 = GhostCellsGenerator(Input=mergeBlocks1)

# create a new 'Calculator'
calculator1 = Calculator(Input=ghostCellsGenerator1)
calculator1.ResultArrayName = 'coordsR'
calculator1.Function = '(coordsX^2+coordsY^2)^.5'

# create a new 'Calculator'
calculator2 = Calculator(Input=calculator1)
calculator2.ResultArrayName = 'z'
calculator2.Function = 'coordsZ'

# Properties modified on calculator2
calculator2.ResultArrayName = 'z'
calculator2.Function = 'coordsZ'

# create a new 'Point Data to Cell Data'
pointDatatoCellData1 = PointDatatoCellData(Input=calculator2)
pointDatatoCellData1.PointDataArraytoprocess = ['coordsR', 'z']


# create a new 'Programmable Filter'
programmableFilter1 = ProgrammableFilter(Input=pointDatatoCellData1)
programmableFilter1.Script = ''
programmableFilter1.RequestInformationScript = ''
programmableFilter1.RequestUpdateExtentScript = ''
programmableFilter1.PythonPath = ''

# set active source
SetActiveSource(pointDatatoCellData1)

# set active source
SetActiveSource(programmableFilter1)

# Properties modified on programmableFilter1
programmableFilter1.Script = """r = inputs[0].CellData['coordsR']
z = inputs[0].CellData['z']
m = (inputs[0].CellData['d00']*inputs[0].CellData['vol00'] +
     inputs[0].CellData['d01']*inputs[0].CellData['vol01'] +
     inputs[0].CellData['d02']*inputs[0].CellData['vol02'] +
     inputs[0].CellData['d03']*inputs[0].CellData['vol03'] +
     inputs[0].CellData['d04']*inputs[0].CellData['vol04'])
v = (inputs[0].CellData['vol00'] +
     inputs[0].CellData['vol01'] +
     inputs[0].CellData['vol02'] +
     inputs[0].CellData['vol03'] +
     inputs[0].CellData['vol04'])
h = 0.1
h2 = h**2
unsorted_out = 0*r
sortargs = argsort(r)
unsortargs = argsort(sortargs)
r_sorted = r[sortargs]
m_sorted = m[sortargs]
z_sorted = z[sortargs]
v_sorted = v[sortargs]
for ind in range(len(r_sorted)):
  r0 = r_sorted[ind]; z0 = z_sorted[ind]
  min, max = searchsorted(r_sorted, (r0-h, r0+h))
  x2 = (r_sorted[min:max] - r0)**2
  y2 = (z_sorted[min:max] - z0)**2
  Psi = exp(1/(x2-h2))*exp(1/(y2-h2))
  unsorted_out[ind] = (sum((Psi*m_sorted[min:max])[y2<h2])/
                       sum((Psi*v_sorted[min:max])[y2<h2]))
output.CellData.append(unsorted_out[unsortargs], 'denbar')
"""
programmableFilter1.RequestInformationScript = ''
programmableFilter1.RequestUpdateExtentScript = ''
programmableFilter1.PythonPath = ''
```